SEEAD: A Semantic-based Approach for Automatic Binary Code De-obfuscation

Zhanyong Tang†, Lei Wang†, Kaiyuan Kuang†, Chao Xue†, Xiaoqing Gong†*
Xiaojiang Chen†, Dingyi Fang†, Zheng Wang‡*
†School of Information Science and Technology, Northwest University, P.R. China.
‡School of Computing and Communications, Lancaster University, UK

***Abstract*—Increasingly sophisticated code obfuscation techniques are quickly adopted by malware developers to escape from malware detection and to thwart the reverse engineering effort of security analysts. State-of-the-art de-obfuscation approaches rely on dynamic analysis, but face the challenge of low code coverage as not all software execution paths and behavior will be exposed at specific profiling runs. As a result, these approaches often fail to discover hidden malicious patterns. This paper introduces SEEAD, a novel and generic semantic-based de-obfuscation system. When building SEEAD, we try to rely on as few assumptions about the structure of the obfuscation tool as possible, so that the system can keep peace with the fast evolving code obfuscation techniques. To increase the code coverage, SEEAD dynamically directs the target program to execute different paths across different runs. This dynamic profiling scheme is rife with taint and control dependence analysis to reduce the search overhead, and a carefully designed protection scheme to bring the program to an error free status should any error happens during dynamic profile runs. As a result, the increased code coverage enables us to uncover hidden malicious behaviors that are not detected by traditional dynamic analysis based de-obfuscation approaches. We evaluate SEEAD on a range of benign and malicious obfuscated programs. Our experimental results show that SEEAD is able to successfully recover the original logic from obfuscated binaries.**

***Index Terms*—Malware Analysis, De-obfuscation, Multiple Execution Paths Exploration**

I. INTRODUCTION

Code obfuscation [1] methods like control flow flattening, garbage code insertion, instruction deformation, binary code encryption and packing [2], and virtualization obfuscation [3], are now commonplace in malware. These code obfuscation techniques make it more difficult to uncover the true logic of the program, giving security analyst an incredibly hard time. Most existing de-obfuscation approaches [4], [5] only target a limited set of specific obfuscation techniques. They work under the assumption that security analysts have priori knowledge of the structure of obfuscation tools (obfuscators) used by the malware developer. This means that these approaches require heavily human involvement (which often takes a lot of time and effort) and can only be applied to known obfuscation methods.

The work presented by Coogan et al. [7] is among the first attempts to automate malware code de-obfuscation without human involvement. Their approach does not require security analysts to manually analyze and identify the obfuscation techniques used by the malware. As a result, the time spent in malware analysis is reduced greatly. While promising, Coogan's method only can deal with malware that uses virtulization-based obfuscation tools such as VMProtect [8] and Virtualizer [9].

In this work, we aim to extend the reach of existing malware de-obfuscation techniques. We present SEEAD, a novel and generic automated code de-obfuscation system. SEEAD is a semantic-based de-obfuscation approach. It makes few assumptions about the structure of obfuscators. Therefore, SEEAD can be applied to existing and unknown obfuscation methods. SEEAD works by first identifying the semantically relevant instructions with dynamic taint analysis and control dependency analysis, and then simplifying the instruction traces of the target binary with these analysis results. Because the whole de-obfuscation process of SEEAD does not require any human involvement, it significantly reduces the time spent in malware analysis.

Similar to most de-obfuscation approaches [10], [11], SEEAD also uses dynamic analysis to characterize the program behavior. However, profiling based dynamic analysis suffers from poor code coverage because the program execution path during profiling runs only represents the application behavior for a given set of inputs. As a result, existing dynamic analysis based de-obfuscation techniques can miss some of the malware behaviors that are only triggered under specific cases (e.g., when a particular file is present, or when a certain command is received). Our approach to the problem is to dynamically adjust the program control logic to direct the program to execute different paths during different profiling runs to increase the code coverage. Our carefully designed recovery scheme ensures that the program can roll back to an error free status if the logic change leads to invalid program execution or corrupted data. To reduce the search space and profiling overhead, we combine taint and control dependence analysis to only change execution branches that dependent on the program input and ignore those do not. As a result, our scheme achieves higher code coverage with reasonable overhead compared to the state-of-the-art dynamic analysis based approaches. The increase code coverage allows us to uncover more hidden malware behaviors.

We have evaluated SEEAD with a range of benign and

*Corresponding authors:
Xiaoqing Gong, Email address: gxq@nwu.edu.cn
Zheng Wang, Email address: z.wang@lancaster.ac.uk

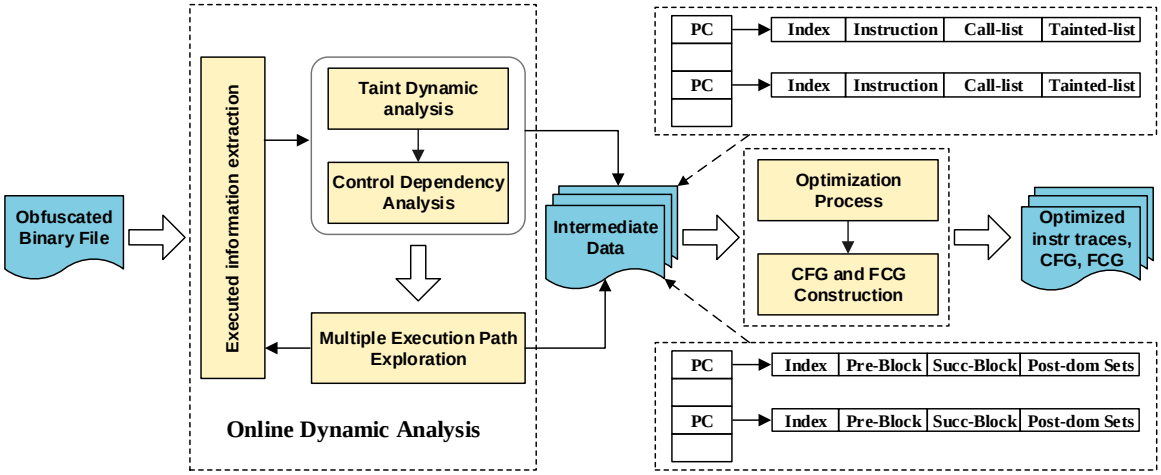

Figure 1: Overview of SEEAD: The figure shows the framework of SEEAD, the components in yellow are the key functions of SEEAD, the components in blue are the input and output of SEEAD. Others are some details produced during the de-obfuscation process.

malicious obfuscated binary programs. Experimental results show that SEEAD is able to eliminate on average 76.8% of the obfuscation instructions and 87.4% of obfuscation instructions generated using virtualization-based obfuscation techniques.

The main contributions of this work are:

- We present SEEAD, a novel and generic semantic-based automated code de-obfuscation system that can apply to unknown obfuscation methods without any human involvement;
- SEEAD is a low-cost solution but provides wider code coverage compared to state-of-the-art dynamic analysis based code de-obfuscation tools;
- Our evaluation performed on a range of benign and malicious obfuscated binaries show that SEEAD is effective at removing obfuscation instructions.

II. SEEAD OVERVIEW

SEEAD is a generic and semantic-based de-obfuscation system. An overview of SEEAD is presented in Figure 1. The input of SEEAD is the obfuscated binary file, the output are the simplified instruction traces, CFG and FCG, which can be easily analyzed and understood.

To perform the code de-obfuscation, SEEAD goes through the following steps:

- Extract the executed information of the obfuscated file based on the dynamic binary instrumentation tool.
- Identify the semantically relevant instructions with dynamic taint analysis and the control dependency analysis in Section 4.
- Present a low-cost solution for exploring multiple execution paths in order to increase the code coverage in Section 5.
- Perform the inter-block and intra-block optimization respectively in Section 6, moreover, SEEAD constructs the CFG and FCG for the optimized instruction traces.

In order to identify the semantically relevant instructions, we need to extract the executed information (i.e., assembly instruction traces, values of registers and memory) of the obfuscated binary file. It is difficult for the common debuggers (e.g., Ollydbg [13], IDA Pro [14]) to handle packing and obfuscated malware, because malware developers usually use various anti-reverse engineering strategies to increase the difficulty of malware analysis. In addition, the cost of analysis is not very optimistic. Dynamic binary instrumentation tools are effective against these anti-reverse engineering strategies. Thus, we build SEEAD on the top of a dynamic binary instrumentation tool called PIN.

III. SEMANTICALLY RELEVANT INSTRUCTION IDENTIFICATION

In this paper, we use dynamic taint analysis to identify values obtained through input operations and instructions influenced by these input-tainted values directly and indirectly. The computation only can capture the explicit information flow from inputs to outputs of the program, but does not consider the implicit information flow [15], it is possible that some behaviors will be missed and the semantics of the program will be changed. To this end, we combine with the explicit data dependencies identified earlier to capture implicit as well as explicit information flow from inputs to outputs.

A. Dynamic Taint analysis

Dynamic taint analysis [16] is widely applied to program security analysis. The basic idea of dynamic taint analysis is to mark the users' sensitive data or untrusted input data as the taint source and track the taint source's propagation path during the executed process.

Similar to prior work [17], SEEAD uses a one-bit tag (0 for "untaint" data and 1 for "taint" data) for each value in memory or general registers in the taint propagation process. If necessary, the one-bit tag can be easily extended to a multiple-bit tag for each value.

At the beginning of the taint propagation process, all tags are assigned to 0. Based on the taint propagation policy, taint

sources (e.g. data read from the network or standard input) will be tagged with 1 as "taint". As program executes, the dynamic taint scheduler propagates the tag information from one instruction to another. It does this by dynamically tracing instructions with information flow. Some other data may be tagged with 1 via information flow. Of course, "taint" data can become "untaint" if its value is reassigned from some safe data.

B. Control Dependency Analysis

Control dependency analysis is mainly used to capture the implicit information flow during the program execution process. For two instructions I and J of a program, J is said to be control-dependent on I if the outcome of I determines if J is executed. More formally, J is control dependent on I only if there is a non-empty path from I to J such that J post-dominates each instruction in this non-empty path except I [18]. The computation of control dependencies has been well-studied in the compiler literature [18].

Algorithm 1 Computing Control Dependencies

Input: An initial tainted instruction trace T
Output: The instruction trace T with control dependencies between instructions identified

1: Construct an initial control flow graph G of trace T;
2: Compute the post-dominator relationships of G;
3: Use post-dominator relationships to compute explicit control dependencies:
4: (a)TaintC = the set of input-tainted conditional control transfers; and
5: (b)DepIns = $\{x \; j \; \exists \; C \in$ TaintC : x control dependent on $C\}$;
6: **while** $\exists$ an indirect control transfer Ins dependent on some $x \in$ DepIns **do**
7: TaintBB $\leftarrow$ basic block of Ins in G;
8: Mark TaintBB as dependent on the direct control transfer in C that x is dependent on;
9: **end while**

We consider two types of control flows: explicit and implicit. Explicit control flows are those conditional control transfers where the predicate is explicitly reflected in the instruction of control transfer. Here, we can use post-dominators to compute explicit control dependencies directly. Implicit control flows are those indirect control transfers such as 'jmp [eax]' where the register eax is data-dependent on the taint sources directly or indirectly. The control dependencies computing algorithm we take is shown in Algorithm 1.

Figure 2 shows an example of two types of control flows: explicit and implicit. The target of conditional control transfer depends on which path is taken on L3, therefore, the instruction L4 and L6 are explicit control dependent on L3. Moreover, the value of register ecx is data dependent on the conditional jump L3, so the target of the indirect control transfer L7 also depends on which path is taken on L3, which is implicit control dependencies.

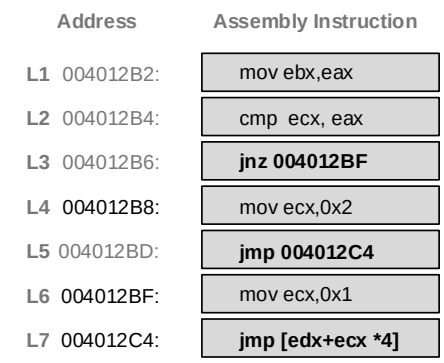

Figure 2: An example of control dependencies: Instruction L4 and L6 are explicit control dependent on L3. Instruction L7 is implicit control dependent on L3.

IV. Multiple Execution Paths Exploration

Traditional dynamic analysis techniques usually only consider a single execution path which typically represent partial program behavior. This practice leads to low code coverage, as some hidden malicious behavior may be missed. To address this problem, previous works usually explore multiple execution paths which depend on some profiling information(e.g. source code, testing information, etc.). However, in the case of malware, we usually do not have access to these profiling information. Moreover, even when the profiling information is available, existing techniques incur higher overhead.

To this end, SEEAD presents a low-cost solution for multiple execution paths exploration. The basic idea is to force the binary to execute requiring no profiling information that We extended the analysis tool with the capability to explore multiple execution paths. It is a great challenge as the search space of all possible paths is usually very large for real world binaries. We know that it is not necessary to obtain all the predicates, because the branch outcomes are usually not affected by any input. Therefore, we use the results of dynamic taint analysis and control dependency analysis to reduce the search space of the predicates. SEEAD only need to consider the tainted branch blocks. The overhead of multiple execution paths exploration will be reduced greatly.

A. Path Exploration

When a conditional branch occurs during the execution of the program, if the current basic block is marked as a taint, we will store the current process address space, then the program will continue executing normally. When the process wishes to terminate later, it replaces the current process address space with the saved snapshot automatically. Here we need to modify the outcome of the decision such that the process continues its execution along the other branch. Of course, there are a lot of branches in the program. In this case, the execution space is explored by selecting next snapshot in a depth-first order. This technique enables us to automatically extract a more complete view of the program.

Here, we will show how to explore multiple execution paths of a program in Figure 3. Assume that the block sequence of

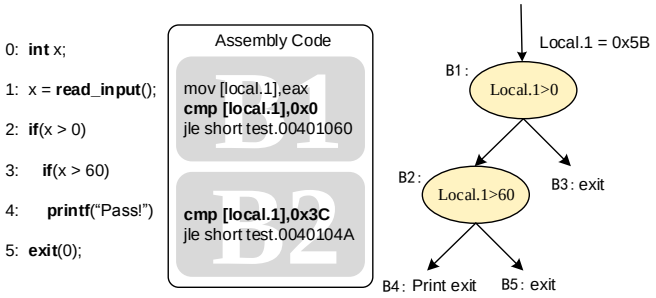

Figure 3: An example of multiple execution paths exploration

the first execution as usual by an arbitrary input is B1, B2, B4, the blocks saved in snapshot list are B2, B1. When the current process wishes to terminate, we replace the current process address space with the saved snapshot B2 firstly, and then B1.

B. Exception Recovery Mechanism

As we all know, the process runs normally until it exits normally or an exception happens. However, SEEAD does not allow the process to terminate. Because the operating system will remove the process-related entries and free its memory, we would not recover the current image to a saved snapshot. Moreover, The program input is merely to allow the execution to preceed, not drive the execution along different paths. It is possible to cause exceptions because of the incorrect input. Thus, We adapt the exception recovery mechanism to prevent any exceptions.

In SEEAD, the obfuscated binary program is first executed as usual by providing arbitrary input. Its recovery mechanism prevents the program from termination. For the program which exit normally, we hook the system API function NtTerminateProcess() of ntdll.dl library to monitor whether the process wishes to terminate. Similarly, for the program crashes, we hook the system API function KiUserExceptionDispatcher() of ntdll.dll library. Whenever the process invokes the API, we can know that a program exception occurs. In this case, if there are unexplored paths left, we will revert the program's current image to a previous state.

V. Optimization Process

The optimization process is mainly divided into two parts: inter-block and intra-block optimization. For the inter-block optimization, we discard those blocks without taint marked which are semantically irrelevant. For the intra-block optimization, we make assumptions as few as possible about the structure of obfuscators. Thus, we present a set of general but simpler semantics-preserving transformations as following:

- **Stack optimization.** There are two cases: a useless push-pop couple and an element A is pushed onto the stack and then popped into an element B.
- **Dead code removal:** Dead code are the instructions whose execution does not modify programs final states or control flow. Every instruction of a block in which all taints get overwritten before being used.
- **Invalid instruction combination:** Invalid instruction combination is some instructions in the combination of which functionally invalided or can cancel each other out. (e.g., add eax, 0xF; sub eax, 0xF).

In order to better analyze and understand the logic of the original program, we construct the CFG and FCG for the optimized results.

CFG and FCG Construction: Construction of CFG and FCG is a basic and highly challenging task for obfuscated binaries, especially for the identification of indirect jump targets and API identification. Since SEEAD is based on dynamic analysis, the targets of indirect jump instructions are precise address which we obtained after dynamic computation. However, for API identification, there is no standard approach in the literature.

As we all know, system calls play an crucial role in malware detection. To some extent, API function sequence is a special representation of malware behavior. Thus, in order to prevent security analysts from extracting the API call sequence and analyzing the program behavior, malware developers usually use various API protection techniques to obfuscate system calls. To construct FCG, we have to develop API identification techniques against API obfuscation to reveal the information of API calls (e.g., address of API calls and their details).

Common API obfuscation techniques can be roughly classified into import table encryption, Hook API and API rewriting. The first two techniques are ineffective to dynamic analysis, because the entry point of the API function can always be traced in dynamic execution process. However, it is challenging to reveal API sequence from the program obfuscated by API rewriting technique. API writing usually copies the first few instructions of the API function to the user space to execute, so we cannot easily identify the entry point of the API function during the execution process. In this paper, we combine code injection and API hook to monitor the API calls and record their invocation information. Finally, we use these collected information of API calls to construct CFG. Since these two tare standard techniques, we omit their details.

VI. Effectiveness Evaluation

A. Effectiveness Analysis

In this subsection, we demonstrate the effectiveness of our de-obfuscation approach by elaborating on the security analysts have only negligible probability of getting the same results with our de-obfuscation approach.

Let t_{ins} denotes the average time of analyzing an instruction, let N_{obf} and N_{simp} denote the instruction number of the obfuscated program and the simplified traces respectively. P_{time} measures how much time we have been able to save when analyzing an obfuscated program. It is defined as:

$$P_{time} = 1 - \frac{N_{simp} \times t_{ins}}{N_{obf} \times t_{ins}} = \frac{N_{obf} - N_{simp}}{N_{obf}} \quad (1)$$

For N_{obf} instructions of obfuscated program, if we want to simplify them into N_{simp} instructions, this yields a total of $\frac{N_{obf}!}{(N_{obf}-N_{simp})!}$ combinations. The security analysts' probability

Table I: Results for programs obfuscated with CF Obfuscator

Samples	Original trace size	Obfuscated trace size	Simplified trace size	Total basic blocks	Input-taint basic blocks	Simplification Score	Difference Score
bin_search	166	221	108	21	19	0.511312	0.3494
bubble_sort	316	641	263	22	6	0.589704	0.16772
huffman	4367	7226	833	59	31	0.884722	0.80925
matrix-mult	651	936	479	44	28	0.488248	0.26421
fibonacci	2930	2950	781	20	12	0.735254	0.73345
factorial	132	174	39	13	10	0.775862	0.70455

Table II: Results for programs obfuscated with MEMP

Samples	Original trace size	Obfuscated trace size	Simplified trace size	Total basic blocks	Input-taint basic blocks	Simplification Score	Difference Score
bin_search	166	1325	549	100	70	0.58566	2.307229
bubble_sort	316	4529	2054	123	96	0.546478	5.5
huffman	4367	34410	10532	170	125	0.693926	1.411724
matrix-mult	651	8230	3758	317	271	0.543378	4.772657
fibonacci	2930	2999	800	29	17	0.733244	0.72696
factorial	132	255	53	26	17	0.792157	0.59848

Table III: Results for programs obfuscated with VMprotect

Samples	Original trace size	Obfuscated trace size	Simplified trace size	Total basic blocks	Input-taint basic blocks	Simplification Score	Difference Score
bin_search	166	859226	215148	314	220	0.749603	1295.072
bubble_sort	316	2371635	501189	215	143	0.788674	1585.401
huffman	4367	5682255	1949412	355	328	0.65693	445.3962
matrix-mult	651	2762309	520705	327	251	0.811496	798.8541
fibonacci	2930	26549	2391	28	26	0.90994	0.18396
factorial	132	12611	1037	25	24	0.91777	6.856061

Table IV: Results for programs obfuscated with Code Virtualizer

Samples	Original trace size	Obfuscated trace size	Simplified trace size	Total basic blocks	Input-taint basic blocks	Simplification Score	Difference Score
bin_search	166	163599	2819	356	52	0.982769	15.98193
bubble_sort	316	605079	5872	322	25	0.990295	17.58228
huffman	4367	2553216	158286	369	114	0.938005	35.24594
matrix-mult	651	693674	50283	315	72	0.927512	76.23963
fibonacci	2930	24818	1647	283	62	0.933637	0.43788
factorial	132	12858	1596	271	96	0.875875	11.09091

of correctly getting these N_{simp} instructions is $\frac{(N_{obf}-N_{simp})!}{N_{obf}!}$. For a 1536B ($N_{obf}$=751 instructions) obfuscated program, the instructions can be simplified as 31 instructions after calculating of our de-obfuscation approach, the security analysts' probability of getting the same results with our de-obfuscation approach therefore is:

$$P\left[Analysis\right] = \frac{(N_{obf} - N_{simp})!}{N_{obf}!} = \frac{(751-31)!}{751!} = 9.68^{-87}$$

The time we have been able to save when analyzing this obfuscated program is:

$$P_{time} = \frac{N_{obf} - N_{simp}}{N_{obf}} = \frac{751-31}{751} = 95.872\%$$

B. Experimental Results

We have implemented a prototype of SEEAD which is implemented in PIN. It supports WIN32 executables. In this section, we present the results of evaluating SEEAD with six samples obfuscated by four obfuscation tools respectively and demonstrate the effectiveness of our approach on multiple execution paths exploration.

Existing virtualization de-obfuscation techniques first reverse engineer the structure of the virtual interpreter; calculate all the byte code instructions based on this information; finally, recover the logic embedded in the virtual interpreter. This approach is very effective when the interpreter structure we dealt with meets the certain needs. However, without the assumption on known interpreter structure, it may not work well.

VMprotect and Code Virtualizer are two representative obfuscation tools that have been considered in previous work [7], [5]. However, these researchers usually do not discuss these non-virtualization obfuscations (e.g., control flow flattening, instruction deformation, encryption, etc.) so we do not know whether they are also able to handle the program obfuscated by these non-virtulaization obfuscations. As far as we know, none of existing approaches on de-obfuscation can be applied to most obfuscation techniques. Thus, we present SEEAD which is effective for most obfuscation techniques. In this subsection, we demonstrate the power of SEEAD with four common obfuscation tools: CF Obfuscator, MEMP [19], Code Virtualizer (CV) [20] and VMprotect (VMP) [21]. CF

Table V: Evaluation of multiple execution paths exploration

Samples	CF Obfuscator				MEMP			
	Branch blocks	Input-taint branch blocks	Dynamic analysis	SEEAD	Branch blocks	Input-taint branch blocks	Dynamic analysis	SEEAD
bin_search	5	5	203	221	5	5	1301	1325
bubble_sort	3	2	641	641	3	2	4529	4529
huffman	15	9	4383	7226	12	9	27325	34410
matrix_mult	15	12	840	936	9	7	7119	8230
fibonacci	4	2	2943	2950	4	2	2992	2999
factorial	2	2	141	174	2	1	189	255

Obfuscator is a binary control flow flattening tool which realized by control flow algorithm OBFWHKD [22]. MEPE combines equivalent deformation, control flow obfuscation and dynamic encryption and decryption. We present the results of evaluating SEEAD with six programs which from [20]. Because the author do not discuss these non-virtulazation obfuscations, we obfuscate these six programs with the CF Obfuscator and MEMP.

Let N_{orig}, N_{obf} and N_{simp} denote the number of instructions for the original program, the obfuscated program and the simplified traces respectively. The simplification score measures how much obfuscation code we have been able to eliminate. It is defined as:

$$Simplification\ Score = \frac{N_{obf} - N_{simp}}{N_{obf}} \quad (2)$$

The difference score measures the instruction number difference between the original program and the simplified traces. It is defined as:

$$Difference\ Score = \frac{|N_{orig} - N_{simp}|}{N_{orig}} \quad (3)$$

The analysis results of programs obfuscated with these four obfuscation tools are in Table I, Table II, Table III and Table IV respectively. The first column shows the name of samples. As shown in the next 3 columns, we report the number of instructions for original program, obfuscated program and simplified traces. The next two columns show the number of total blocks and input-taint blocks respectively. Finally, we present the simplification score and in the last two columns.

Figure 4 shows the comparison results of simplification scores in all samples. The simplification score introduced by MEMP is on average about 0.65, which means that SEEAD is able to eliminate about 65% of obfuscation instructions introduced by CF Obfuscator. The simplification score introduced by CV is over 0.94 on average. The simplification scores introduced by CF Obfuscator and VMP lie in the middle. They are about 0.67 and 0.81 on average respectively.

Similarity, the comparison results of difference scores are shown in Figure 5. The highest difference score is about 689 on average which is introduced by VMP, and the lowest score is over 0.5 on average which is introduced by CF Obfuscator. The difference scores introduced by MEMP and CV lie in the middle. They are about 2.6 and 26 on average respectively.

Overall, these results are encouraging, especially for virtualization obfuscations, as SEEAD only identifies those instructions that are semantically relevant with the original code, and discards those that are semantically irrelevant. Our evaluation results show that we can straightforwardly reconstruct the logic of original program and analyze them correctly with the functionality we have traced.

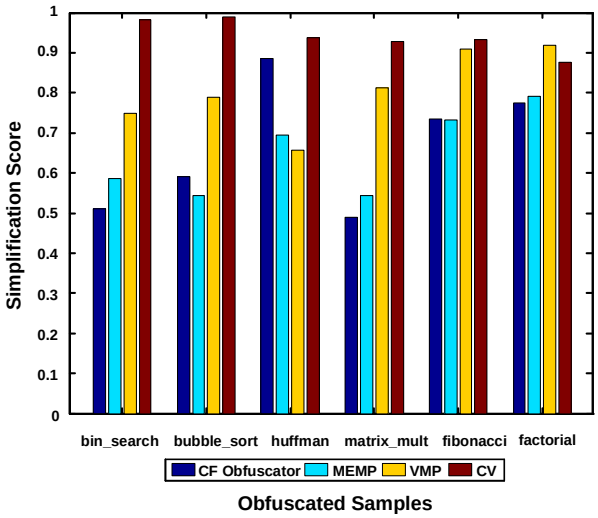

Figure 4: Comparison results of simplification scores

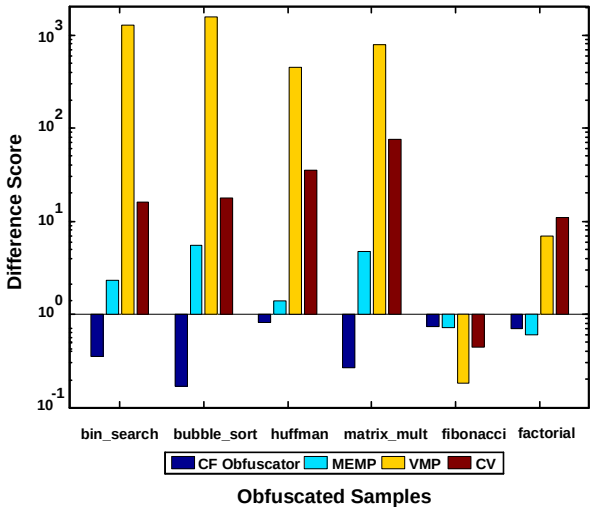

Figure 5: Comparison results of difference Scores

We observed that most of the "missed" instructions were classified into two categories: on the one hand, instructions which performed some preparation work like allocating memory or initializing data structures; on the other hand, instructions which performed some invalid actions that were semanti-

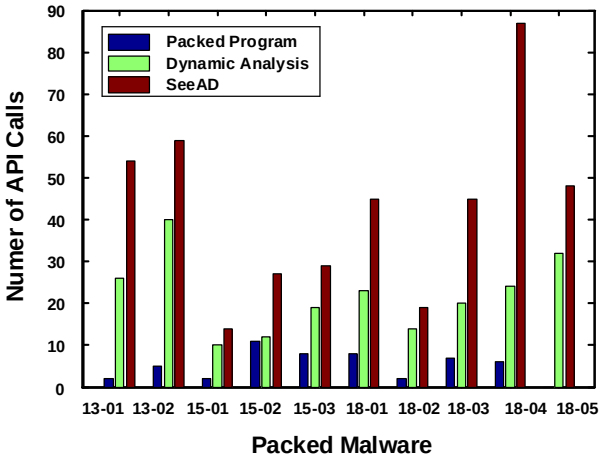

Figure 6: Comparison results of API calls

cally irrelevant such as garbage instructions, invalid instruction combination. All of these instructions were identified by our analysis.

We examined the results by hand, and found the reason for the higher simplification scores is that all the test cases we used are all toy programs. We believe that this paper is just an initial step on developing an advanced and functionally powerful de-obfuscation tool.

The results in Table I, Table II, Table III and Table IV show that the extraordinary increase in the number of executed instructions for these four obfuscator tools. For example, bin_search executes 166 instructions in the original program. However, the number of executed instructions of the program obfuscated by CF Obfuscator, MEMP, CV and VMP are 221, 1325, 163599 and 659226 respectively.

As we all know, traditional dynamic analysis typically represents partial program behavior and the coverage heavily relies on good inputs which may not be available. We compare the results of SEEAD with traditional dynamic analysis and demonstrate the effectiveness of our approach on multiple execution paths exploration. The comparison results are presented in Table V. Columns 2-5 present the comparison results of the program obfuscated by CF Obfuscator. Column 2 presents the number of the branch blocks. The number of branch blocks influenced by the input-tainted values is shown in column 3. We explore multiple execution paths according to these input-taint branch blocks. Columns 4-5 present the instructions that are covered by different approaches. Particularly, column 4 shows the number of instructions that are executed by traditional dynamic analysis. Column 5 shows those extracted by SEEAD. Similarly, columns 6-9 present the comparison results of the program obfuscated by MEMP.

Figure 6 shows the comparison results between traditional analysis and SEEAD. From the coverage data, we observed that SEEAD could cover more instructions than dynamic analysis, however, for our test cases, the increase in the number of instructions was less. We examined the results by hand, and found two reasons. First, we provided the good inputs for the test cases in the dynamic analysis, so it can cover most instructions. Second, the increase in the number of instructions was closely related to the function of the obfuscated code. In general,from the experimental results, we can ensure that SEEAD can be used to handle most obfuscations without any human involvement and at the same time increase the code coverage. The effectiveness and efficiency of malware analysis will be improved by SEEAD.

VII. Discussion and Future Work

Existing binary an be roughly classified into static [23], dynamic [24], and symbolic analysis [10], [11]. However, all of three techniques have their limitations. Now, we compare these techniques in terms of code coverage, the capability of handling packing and obfuscation and scalability. Static analysis usually has good code coverage, and which is very scalable. However, it is difficult for static analysis to handle packing and obfuscated program, because some instructions of the target binary are dynamic computing. For dynamic analysis, it usually produces only partial program behavior and the code coverage sometimes heavily relies on good inputs which may not be available. For symbolic analysis, it is able to construct inputs with the path conditions, but has difficulty in handling packed or obfuscated binaries.

It is difficult for SEEAD to model multiple threads into a single execution since their execution sequence is nondeterministic. X-Force [25] adopts a simple and effective approach to serialize the execution of threads. The calls to thread creation library functions are replaced with direct function calls to the starting functions of threads, which avoid creating multiple threads and guarantees code coverage at the same time. However, it is ineffective to analyze the behavior which is sensitive to schedules. In the future we will explore handling the real concurrent executions.

VIII. Related Work

a) De-obfuscation mechanisms.: De-obfuscation is not a new problem, thus, a number of solutions already exist. Udupa et al. [26] discuss the deobfuscating code that has been obfuscated by control flow flattening [27], which resembles emulation-based obfuscation in some ways. Jones et al. [28] describe a technique for specializing away interpretive code. These works are based on static, which are ineffective against complex obfuscated binaries, e.g., due to dynamic encryption and decryption and and self-modifying code.

Sharif et al. represent an approach [5] for de-obfuscation, it first reverse engineers the VM emulator, and then use the information to work out individual byte code instructions. However, the proposed approach may not work well when the emulator uses techniques that do not fit these assumptions.There is a semantic-based approach for de-obfuscation. Coogan et al. [7] uses equational reasoning about assembly instruction semantics to simplify the obfuscation code from execution traces of virtualization obfuscated programs. It does not seem strightforward to control the whole e-obfuscation process to recover the logic of the program. Moreover, this paper does

not construct the CFG and FCG for better understanding the de-obfuscated results.

b) Multiple Execution Paths Exploration.: Early approaches on multiple execution paths exploration usually rely on profiling information to construct concrete program inputs [29], [30], such as source code, software testing and debugging information. Unfortunately, in practice, such information is generally not available. Hence, for malware, the assumption can be considered unrealistic. In particular, the work in [12] requires concrete inputs firstly and then mutate such inputs to explore different paths which incurs high overhead.

There is an approach for multiple execution paths exploration in [25] by forcing the branch outcomes to be reversed to construct control flow graphs, However, partial paths they explored are infeasible. Similar techniques are proposed to expose hidden behavior in Android apps [31], [32]. These techniques randomly determine each branch's outcome, facing the challenge of excessive infeasible.

IX. Conclusions

This paper has presented SEEAD, a novel, generic framework for code de-obfuscation, targeting malware detection. SEEAD employs dynamic taint analysis and control dependency analysis to carefully direct the program execution path across profiling runs to increase the code coverage. It then simplifies the instruction traces of the target binary to perform code de-obfuscation. SEEAD is fully automatic and requires little human involvement. We evaluate SEEAD on a range of benign and malicious obfuscated programs. Experimental results show that SEEAD can successfully recover the original logic from obfuscated binaries.

References

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[2] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[3] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 94–109.

[4] R. Rolles, "Unpacking virtualization obfuscators," in *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[5] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM, 2007, pp. 46–53.

[6] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 275–284.

[7] "Vmprotect - new-generation software protection," http://www.vmprotect.ru/, Tech. Rep.

[8] "Code virtualizer: Total obfuscation against reverse engineering," Oreans Technologies, http://www.oreans.com/codevirtualizer.php, Tech. Rep., 2008.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[10] V. Chipounov, V. Kuznetsov, and G. Candea, *S2E: a platform for in-vivo multi-path analysis of software systems*. ACM, 2012, vol. 47, no. 4.

[11] O. Yuschuk, "Ollydbg 1.1: A 32-bit assembler level analysing debugger for microsoft windows, june 2004."

[12] "Ida pro: a windows, linux or mac os x hosted," https://www.hex-rays.com/products/ida/index.shtml, Tech. Rep.

[13] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Acm Sigplan Notices*, vol. 39, no. 11. ACM, 2004, pp. 85–96.

[14] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008, pp. 74–83.

[15] A. Lakhotia and E. U. Kumar, "Abstracting stack to detect obfuscated calls in binaries," in *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 2004, pp. 17–26.

[16] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*. Addison wesley, 1986.

[17] W. H. Fang Dingyi, Li Guanghui, "Research on deformation based binary," *Journal of Sichuan University (Engineering Science Edition)*, 2014,1:003.

[18] "Obfuscated samples," http://www.cs.arizona.edu/projects/lynx/Samples/, Tech. Rep.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[20] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[21] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," DTIC Document, Tech. Rep., 2006.

[22] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 487–498.

[23] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 2014 USENIX Security Symposium, San Diego, CA (August 2014)*, 2014.

[24] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[25] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 2001, pp. 193–202.

[26] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[27] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[28] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas, "Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 38–52.

[29] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 231–245.

[30] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 188–197.

[31] Z. Wang, R. Johnson, R. Murmuria, and A. Stavrou, "Exposing security risks for commercial mobile devices," in *Computer Network Security*. Springer, 2012, pp. 3–21.